**Capítulo 10 - Criptografia de Ponta a Ponta**

Capítulo 9 explicou a segurança de transporte via protocolos como TLS e Noise. Ao mesmo tempo, passei um bom tempo explicando onde a confiança está enraizada na web: centenas de autoridades certificadoras (CAs) confiáveis pelo seu navegador e sistema operacional. Embora não seja perfeito, este sistema tem funcionado até agora para a web, que é uma rede complexa de participantes que nada sabem uns dos outros.

Esse problema de encontrar maneiras de confiar em outros (e em suas chaves públicas) e fazê-lo escalar está no centro da criptografia no mundo real. Um famoso criptógrafo já foi ouvido dizendo: “A criptografia simétrica está resolvida”, para descrever um campo de pesquisa que já havia cumprido seu tempo. E, em grande parte, a afirmação era verdadeira. Raramente temos problemas ao criptografar comunicações, e temos forte confiança nos algoritmos de criptografia atualmente em uso. Quando se trata de criptografia, a maioria dos desafios de engenharia não diz respeito mais aos próprios algoritmos, mas sim a quem são Alice e Bob e como prová-lo.

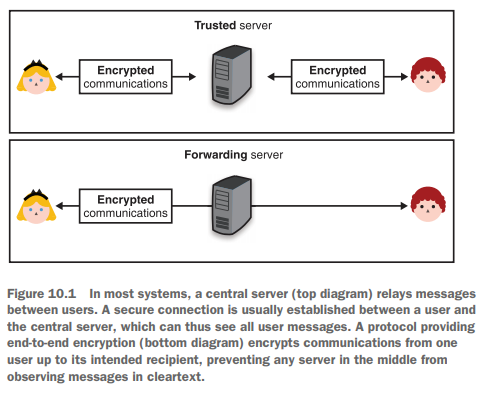
Este capítulo cobre:

* Criptografia de ponta a ponta e sua importância
* Diferentes tentativas de resolver a criptografia de e-mail
* Como a criptografia de ponta a ponta está mudando o cenário da troca de mensagens

**10.1 Por que criptografia de ponta a ponta?**

Este capítulo começa com um “por quê” em vez de um “o quê”. Isso porque criptografia de ponta a ponta é mais um conceito do que um protocolo criptográfico; é um conceito de garantir comunicações entre dois (ou mais) participantes através de um caminho adversarial. Comecei este livro com um exemplo simples: a Rainha Alice queria enviar uma mensagem para Lorde Bob sem que ninguém no meio pudesse vê-la. Hoje em dia, muitos aplicativos como e-mail e mensagens existem para conectar usuários, e a maioria deles raramente criptografa mensagens do início ao fim.

Você pode perguntar: o TLS não é suficiente? Em teoria, poderia ser. Você aprendeu no capítulo 9 que o TLS é usado em muitos lugares para proteger comunicações. Mas a criptografia de ponta a ponta envolve seres humanos reais. Em contraste, o TLS é mais frequentemente usado por sistemas que são “homens-no-meio” por design (ver figura 10.1). Nesses sistemas, o TLS é apenas usado para proteger as comunicações entre um servidor central e seus usuários, permitindo que o servidor veja tudo. Efetivamente, esses servidores MITM ficam entre os usuários, são necessários para o funcionamento do aplicativo e são terceiros confiáveis do protocolo. Ou seja, temos que confiar nessas partes do sistema para que o protocolo seja considerado seguro (alerta de spoiler: não é um ótimo protocolo).



Na prática, existem topologias muito piores. As comunicações entre um usuário e um servidor podem passar por muitos saltos de rede, e alguns desses saltos podem ser máquinas que observam o tráfego (frequentemente chamadas de middleboxes). Mesmo que o tráfego esteja criptografado, algumas middleboxes são configuradas para encerrar a conexão TLS (dizemos que elas terminam o TLS) e ou encaminham o tráfego em claro a partir desse ponto, ou iniciam outra conexão TLS com o próximo salto. A terminação do TLS às vezes é feita por “boas” razões: para filtrar melhor o tráfego, balancear conexões geograficamente ou dentro de um data center, e assim por diante. Isso aumenta a superfície de ataque, pois o tráfego agora está visível em claro em mais lugares. Às vezes, a terminação do TLS é feita por “más” razões: para interceptar, gravar e espionar o tráfego.

Em 2015, a Lenovo foi pega vendendo laptops com CAs personalizados e softwares pré-instalados (abordados no capítulo 9). O software estava interceptando conexões HTTPS usando as CAs da Lenovo e injetando anúncios nas páginas web. Mais preocupante, países como China e Rússia foram pegos redirecionando tráfego na internet, fazendo-o passar por suas redes para interceptar e observar conexões. Em 2013, Edward Snowden vazou um número massivo de documentos da NSA mostrando os abusos de muitos governos (não apenas dos EUA) em espionar comunicações das pessoas interceptando os cabos de internet que conectam o mundo.

Possuir e visualizar dados de usuários também é uma responsabilidade para as empresas. Como mencionei várias vezes neste livro, violações e invasões acontecem com muita frequência e podem ser devastadoras para a credibilidade de uma empresa. Do ponto de vista legal, leis como o Regulamento Geral de Proteção de Dados (GDPR) podem acabar custando muito dinheiro às organizações. Solicitações governamentais como as infames Cartas de Segurança Nacional (NSLs) que às vezes impedem as empresas e pessoas envolvidas de sequer mencionarem que receberam as cartas (as chamadas ordens de silêncio) podem ser vistas como custos e estresses adicionais para uma organização também — a menos que você não tenha muito a compartilhar.

No fim das contas, se você está usando um aplicativo online popular, há grandes chances de que um ou mais governos já tenham acesso ou tenham a capacidade de obter acesso a tudo o que você escreveu ou enviou ali. Dependendo do modelo de ameaça de um aplicativo (o que o aplicativo deseja proteger) ou do modelo de ameaça dos usuários mais vulneráveis de um aplicativo, a criptografia de ponta a ponta desempenha um papel fundamental para garantir a confidencialidade e privacidade dos usuários finais.

Este capítulo aborda diferentes técnicas e protocolos que foram criados para estabelecer confiança entre as pessoas. Em particular, você aprenderá como funciona a criptografia de e-mail atualmente e como as mensagens seguras estão mudando o cenário das comunicações criptografadas de ponta a ponta.

**10.2 Uma raiz de confiança em lugar nenhum**

Um dos cenários mais simples de criptografia de ponta a ponta é o seguinte: Alice quer enviar um arquivo criptografado para Bob pela internet. Com todos os algoritmos criptográficos que você aprendeu nos primeiros capítulos deste livro, provavelmente pode imaginar uma forma de fazer isso. Por exemplo:

1. Bob envia sua chave pública para Alice.
2. Alice criptografa o arquivo com a chave pública de Bob e o envia para Bob.

Talvez Alice e Bob possam se encontrar pessoalmente ou usar outro canal seguro que já compartilham para trocar a chave pública na primeira mensagem. Se isso for possível, dizemos que eles possuem uma maneira fora de banda de criar confiança. Porém, isso nem sempre é o caso. Você pode imaginar eu incluindo minha própria chave pública neste livro e pedindo que você a use para me enviar uma mensagem criptografada para algum endereço de e-mail. Quem garante que meu editor não substituiu a chave pública pela dela?

O mesmo vale para Alice: como ela descobre se a chave pública que recebeu é realmente a de Bob? É possível que alguém no meio tenha adulterado a primeira mensagem. Como você verá neste capítulo, a criptografia não possui uma verdadeira resposta para esse problema de confiança. Em vez disso, ela fornece diferentes soluções para ajudar em cenários distintos. A razão de não haver uma solução única é que estamos tentando conectar a realidade (seres humanos reais) com um protocolo criptográfico teórico.

Todo esse negócio de proteger chaves públicas contra adulteração é o problema mais difícil nas aplicações práticas de criptografia de chave pública. É o “calcanhar de Aquiles” da criptografia de chave pública, e muita complexidade de software está envolvida em resolver este único problema.

—Zimmermann et al. (“PGP User’s Guide Volume I: Essential Topics”, 1992)

Voltando ao nosso cenário simples, onde Alice quer enviar um arquivo para Bob, e assumindo que sua conexão não confiável é tudo o que possuem, eles enfrentam uma espécie de problema impossível de confiança. Alice não tem uma boa maneira de saber com certeza qual é de fato a chave pública de Bob. É um dilema do tipo ovo-e-galinha. Ainda assim, deixe-me apontar que, se não houver um atacante ativo malicioso MITM substituindo a chave pública de Bob na primeira mensagem, então o protocolo está seguro. Mesmo que as mensagens estejam sendo gravadas passivamente, já seria tarde demais para um atacante descriptografar a segunda mensagem posteriormente.

Claro, contar com o fato de que suas chances de ser alvo de um MITM ativo não sejam muito altas não é a melhor forma de abordar a criptografia. Infelizmente, muitas vezes não temos como evitar isso. Por exemplo, o Google Chrome vem com um conjunto de autoridades certificadoras (CAs) que escolhe confiar, mas como você obteve o Chrome inicialmente? Talvez tenha usado o navegador padrão do seu sistema operacional, que confia em seu próprio conjunto de CAs. Mas de onde veio esse conjunto? Do laptop que você comprou. E de onde veio o laptop? Como você pode ver rapidamente, são “tartarugas até lá embaixo”. Em algum ponto, você terá que confiar que algo foi feito corretamente.

Um modelo de ameaça normalmente decide parar de abordar questões após uma tartaruga específica e considera que qualquer tartaruga abaixo está fora do escopo. É por isso que o restante do capítulo assumirá que você tem uma forma segura de obter alguma raiz de confiança. Todos os sistemas baseados em criptografia funcionam baseados em uma raiz de confiança, algo sobre o qual o protocolo pode construir segurança. Uma raiz de confiança pode ser um segredo ou um valor público com o qual começamos o protocolo ou um canal fora de banda que podemos usar para obtê-los.

**10.3 O fracasso do e-mail criptografado**

O e-mail foi criado como (e ainda é hoje) um protocolo não criptografado. Podemos culpar apenas uma época em que a segurança era uma preocupação secundária. A criptografia de e-mail começou a se tornar mais do que uma ideia após o lançamento de uma ferramenta chamada *Pretty Good Privacy* (PGP) em 1991. Na época, o criador do PGP, Phil Zimmermann, decidiu lançar o PGP em reação a um projeto de lei que quase se tornou lei no mesmo ano. O projeto permitiria ao governo dos EUA obter todas as comunicações de voz e texto de qualquer empresa de comunicação eletrônica e fabricante. Em seu ensaio de 1994 "Por que você precisa de PGP?", Philip Zimmermann conclui dizendo:

"O PGP dá poder às pessoas para assumirem o controle de sua própria privacidade. Existe uma necessidade social crescente por isso. É por isso que eu o escrevi."

O protocolo foi finalmente padronizado no RFC 2440 como OpenPGP em 1998 e ganhou força com o lançamento da implementação de código aberto GNU Privacy Guard (GPG), por volta da mesma época. Hoje, o GPG ainda é a principal implementação, e as pessoas usam os termos GPG e PGP de forma intercambiável.

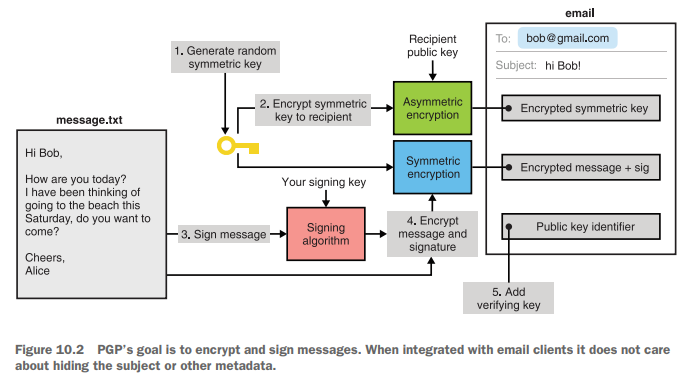
**10.3.1 PGP ou GPG? E como funciona?**

O PGP, ou OpenPGP, funciona simplesmente utilizando criptografia híbrida (tratada no capítulo 6). Os detalhes estão no RFC 4880, a última versão do OpenPGP, e podem ser simplificados nos seguintes passos:

1. O remetente cria um e-mail. Neste ponto, o conteúdo do e-mail é compactado antes de ser criptografado.
2. A implementação do OpenPGP gera uma chave simétrica aleatória e criptografa o e-mail simetricamente usando essa chave.
3. A chave simétrica é criptografada assimetricamente para cada chave pública dos destinatários (usando as técnicas que você aprendeu no capítulo 6).
4. Todas as versões criptografadas da chave simétrica dos destinatários são concatenadas com a mensagem criptografada. O corpo do e-mail é substituído por esse *blob* de dados e enviado a todos os destinatários.
5. Para descriptografar um e-mail, o destinatário usa sua chave privada para descriptografar a chave simétrica e, então, descriptografa o conteúdo do e-mail usando essa chave simétrica.

Observe que o OpenPGP também define como um e-mail pode ser assinado para autenticar o remetente. Para isso, o corpo do e-mail em texto claro é *hasheado* e então assinado com a chave privada do remetente. A assinatura é então adicionada à mensagem antes de ser criptografada no passo 2. Finalmente, para que o destinatário saiba qual chave pública usar para verificar a assinatura, a chave pública do remetente é enviada junto com o e-mail criptografado no passo 4. Eu ilustro o fluxo do PGP na figura 10.2.

**Exercício**  
Você sabe por que o conteúdo do e-mail é compactado antes de ser criptografado e não depois?

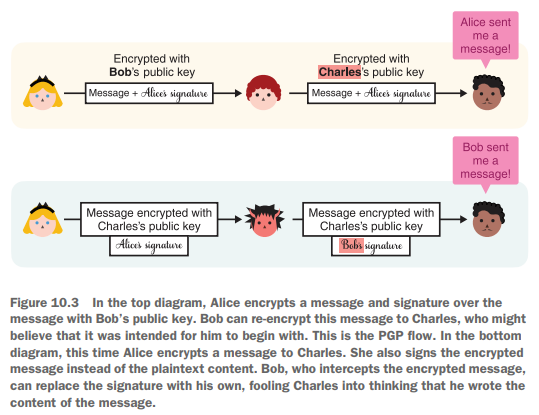
Não há nada intrinsecamente errado com este design à primeira vista. Parece impedir que atacantes MITM vejam o conteúdo do seu e-mail, embora o assunto e outros cabeçalhos de e-mail não sejam criptografados.

**NOTA** É importante observar que a criptografia nem sempre consegue ocultar todos os metadados. Em aplicações conscientes de privacidade, metadados são um grande problema e podem, nos piores casos, desanonimizar você! Por exemplo, em protocolos criptografados de ponta a ponta, pode não ser possível descriptografar mensagens entre usuários, mas provavelmente é possível saber quais são seus endereços IP, o tamanho das mensagens que enviam e recebem, com quem falam com frequência (seus gráficos sociais), e assim por diante. Muito trabalho de engenharia é dedicado a ocultar esse tipo de metadado.

Ainda assim, nos detalhes, o PGP é na verdade bastante ruim. O padrão OpenPGP e sua principal implementação, o GPG, utilizam algoritmos antigos, e a compatibilidade com versões anteriores impede que a situação melhore. A questão mais crítica é que a criptografia não é autenticada, o que significa que qualquer um que intercepte um e-mail não assinado pode ser capaz de adulterar o conteúdo criptografado em certo grau, dependendo do algoritmo de criptografia usado. Por essa razão somente, eu não recomendaria o uso do PGP atualmente.

Uma falha surpreendente do PGP decorre do fato de que as operações de assinatura e criptografia são compostas sem cuidado. Em 2001, Don Davis apontou que, por causa dessa composição ingênua de algoritmos criptográficos, alguém pode recriptografar um e-mail assinado que recebeu e enviá-lo para outro destinatário. Isso efetivamente permite que Bob envie a você o e-mail que Alice enviou a ele como se você fosse o destinatário original!

Se você está se perguntando, assinar o *ciphertext* em vez do *plaintext* ainda é falho, pois alguém poderia simplesmente remover a assinatura que acompanha o *ciphertext* e adicionar sua própria assinatura. Na prática, Bob poderia fingir que enviou um e-mail a você que, na verdade, foi escrito por Alice. Eu recapitulo esses dois problemas de assinatura na figura 10.3.



O toque final é que o algoritmo não oferece *forward secrecy* por padrão. Como lembrete, sem *forward secrecy*, o comprometimento de sua chave privada implica que todos os e-mails anteriores enviados a você e criptografados sob essa chave podem agora ser descriptografados. Você ainda pode forçar *forward secrecy* trocando sua chave PGP, mas esse processo não é simples (você pode, por exemplo, assinar sua nova chave com sua chave antiga), e a maioria dos usuários simplesmente não se dá ao trabalho. Recapitulando, lembre-se que:

* O PGP usa algoritmos criptográficos antigos.
* O PGP não possui criptografia autenticada e, portanto, não é seguro se usado sem assinaturas.
* Devido a um mau design, receber uma mensagem assinada não significa necessariamente que você era o destinatário pretendido.
* Não há *forward secrecy* por padrão.

**Exercício**  
Você consegue pensar em uma forma inequívoca de assinar uma mensagem?

**10.3.2 Escalando confiança entre usuários com a teia de confiança (*web of trust*)**

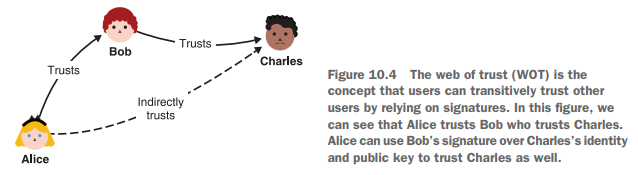
Então, por que estou realmente falando sobre PGP aqui? Bem, há algo interessante sobre o PGP que ainda não mencionei: como você obtém e como pode confiar nas chaves públicas de outras pessoas? A resposta é que, no PGP, você mesmo constrói a confiança!

Ok, o que isso significa? Imagine que você instala o GPG e decide que quer criptografar algumas mensagens para seus amigos. Para começar, você deve primeiro encontrar uma maneira segura de obter as chaves públicas PGP dos seus amigos. Encontrá-los pessoalmente é uma forma segura de fazer isso. Vocês se encontram, copiam as chaves públicas em um pedaço de papel e, em casa, você digita essas chaves no seu laptop. Agora, você pode enviar mensagens assinadas e criptografadas para seus amigos com o OpenPGP. Mas isso é tedioso. Você tem que fazer isso para cada pessoa com quem deseja trocar e-mails? Claro que não. Vamos considerar o seguinte cenário:

* Você obteve a chave pública de Bob pessoalmente e, portanto, confia nela.
* Você não possui a chave pública de Mark, mas Bob possui e confia nela.

Tire um momento para pensar no que você poderia fazer para confiar na chave pública de Mark. Bob pode simplesmente assinar a chave de Mark, mostrando a você que ele confia na associação entre a chave pública e o e-mail de Mark. Se você confia em Bob, agora pode confiar na chave pública de Mark e adicioná-la ao seu repertório. Esta é a ideia principal por trás do conceito de confiança descentralizada do PGP. É chamado de *web of trust* (WOT), como ilustrado na figura 10.4.

Você às vezes verá "festas de assinatura de chaves" em conferências, onde as pessoas se encontram pessoalmente e assinam as respectivas chaves públicas. Mas, na prática, a maioria das pessoas não depende da WOT para expandir seu círculo PGP.



**10.3.3 Descoberta de chaves é um problema real**

O PGP tentou outra forma de resolver o problema de descobrir chaves públicas — registros de chaves. O conceito é bastante simples: publique sua chave pública PGP e as assinaturas associadas de outros que atestam sua identidade em alguma lista pública para que as pessoas possam encontrá-la. Na prática, isso não funciona, já que qualquer um pode publicar uma chave e uma assinatura associada supostamente vinculada ao seu e-mail. De fato, alguns atacantes propositalmente falsificaram chaves em servidores de chaves, embora talvez mais para causar confusão do que para espionar e-mails. Em alguns cenários, podemos relaxar nosso modelo de ameaça e permitir que uma autoridade confiável ateste identidades e chaves públicas. Pense, por exemplo, em uma empresa gerenciando os e-mails de seus funcionários.

Em 1995, a empresa RSA propôs o Secure/Multipurpose Internet Mail Extensions (S/MIME) como uma extensão do formato MIME (que por sua vez é uma extensão do padrão de e-mail) e como uma alternativa ao PGP. O S/MIME, padronizado na RFC 5751, adotou um distanciamento interessante do WOT ao usar uma infraestrutura de chave pública para construir confiança. Essa é praticamente a única diferença conceitual que o S/MIME tem com o PGP. Como as empresas possuem processos para integrar e desligar funcionários, faz sentido que elas comecem a usar protocolos como o S/MIME para impulsionar a confiança em seu ecossistema interno de e-mail. É importante observar que tanto o PGP quanto o S/MIME são geralmente usados ​​sobre o Simple Mail Transfer Protocol (SMTP), que é o protocolo usado hoje para enviar e receber e-mails. O PGP e o S/MIME também foram inventados posteriormente e, por esse motivo, sua integração com clientes SMTP e de e-mail está longe de ser perfeita. Por exemplo, apenas o corpo de um e-mail é criptografado, não o assunto ou qualquer outro cabeçalho. O S/MIME, assim como o PGP, também é um protocolo bastante antigo que utiliza criptografia e práticas desatualizadas. Assim como o PGP, ele não oferece criptografia autenticada. Pesquisas recentes (Efail: “Quebrando a Criptografia de E-mail S/MIME e OpenPGP usando Canais de Exfiltração”) sobre a integração de ambos os protocolos em clientes de e-mail mostraram que a maioria deles era vulnerável a ataques de exfiltração, nos quais um invasor que observa e-mails criptografados pode recuperar o conteúdo enviando versões adulteradas aos destinatários. No final, essas deficiências podem nem importar, já que a maioria dos e-mails enviados e recebidos no mundo circula pela rede global sem criptografia. O PGP provou ser bastante difícil de usar para usuários não técnicos, bem como avançados, que precisam entender as muitas sutilezas e fluxos do PGP para criptografar seus e-mails. Por exemplo, não é incomum ver usuários respondendo a um e-mail criptografado sem usar criptografia, citando a conversa inteira em texto não criptografado. Além disso, o suporte insuficiente (ou inexistente) ao PGP por clientes de e-mail populares não ajudou.

*Na década de 1990, eu estava animado com o futuro e sonhava com um mundo onde todos instalariam o GPG. Agora, ainda estou animado com o futuro, mas sonho com um mundo onde eu possa desinstalá-lo.*

*—Moxie Marlinspike (“GPG and Me,” 2015)*

Por essas razões, o PGP vem perdendo suporte gradativamente (por exemplo, a Golang removeu o suporte para PGP de sua biblioteca padrão em 2019), enquanto cada vez mais aplicações de criptografia do mundo real buscam substituir o PGP e resolver seus problemas de usabilidade. Hoje, é difícil argumentar que a criptografia de e-mails algum dia terá o mesmo nível de sucesso e adoção que, por exemplo, o HTTPS.

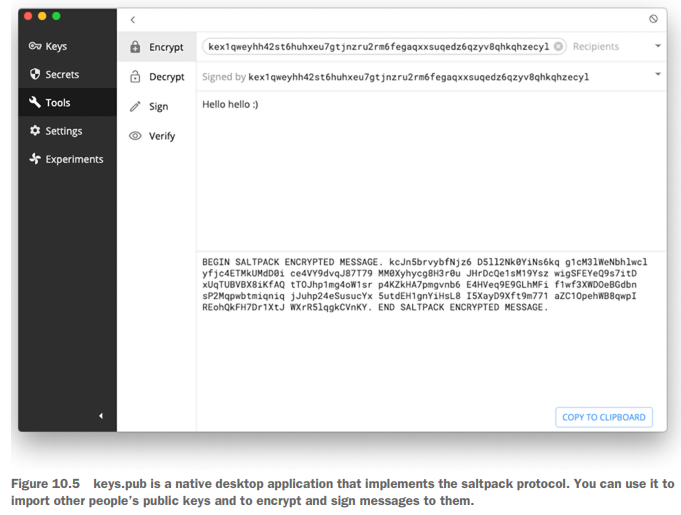
*Se as mensagens puderem ser enviadas em texto simples, elas serão enviadas em texto simples. O e-mail é de ponta a ponta sem criptografia por padrão. A base do correio eletrônico é o texto simples. Todos os softwares de e-mail convencionais esperam texto simples. De maneiras significativas, o sistema de e-mail da internet é simplesmente projetado para não ser criptografado.*

*—Thomas Ptacek (“Stop Using Encrypted Email,” 2020)*

**10.3.4 Se não for PGP, então o quê?**

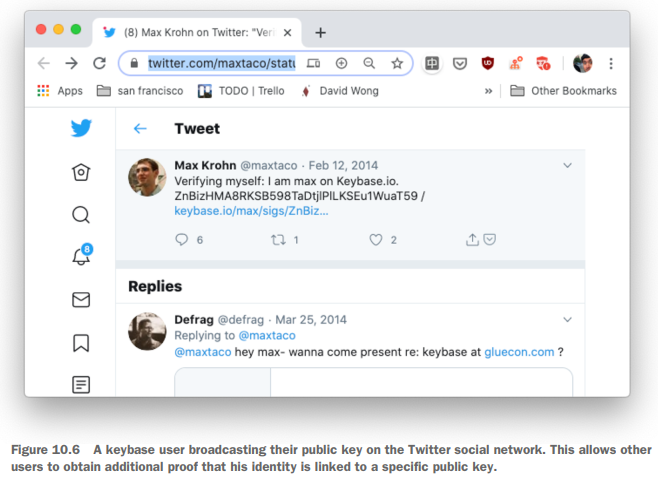
Gastei várias páginas explicando como um design simples como o PGP pode falhar de muitas formas diferentes e surpreendentes na prática. Sim, eu recomendaria não usar o PGP. Embora a criptografia de e-mail ainda seja um problema não resolvido, alternativas estão sendo desenvolvidas para substituir diferentes casos de uso do PGP.

*saltpack* é um protocolo e formato de mensagem semelhante ao PGP. Ele tenta corrigir algumas das falhas do PGP que mencionei. Em 2021, as principais implementações do *saltpack* são o *keybase* (<https://keybase.io>) e o *keys.pub* (<https://keys.pub>). A figura 10.5 ilustra a ferramenta *keys.pub*.



Essas implementações abandonaram o modelo WOT e permitem que os usuários transmitam suas chaves públicas em diferentes redes sociais a fim de vincular sua identidade às suas chaves públicas (como ilustrado na figura 10.6). Obviamente, o PGP não poderia ter antecipado este mecanismo de descoberta de chaves, pois antecede o boom das redes sociais.

Por outro lado, a maioria das comunicações seguras hoje em dia está longe de serem mensagens únicas, e o uso dessas ferramentas é cada vez menos relevante. Na próxima seção, falarei sobre mensagens seguras, um dos campos que visa substituir o aspecto de comunicação do PGP.



**10.4 Mensagens seguras: Uma visão moderna da criptografia de ponta a ponta com o Signal**

Em 2004, o *Off-The-Record* (OTR) foi introduzido em um artigo intitulado *"Off-the-Record Communication, or, Why Not To Use PGP"*. Diferente do PGP ou S/MIME, o OTR não é usado para criptografar e-mails, mas sim mensagens de chat; especificamente, ele estende um protocolo de chat chamado *Extensible Messaging and Presence Protocol* (XMPP).

Uma das características distintas do OTR era a *negabilidade* — uma alegação de que os destinatários de suas mensagens e observadores passivos não podem usar as mensagens que você enviou contra você em um tribunal. Como as mensagens enviadas são autenticadas e criptografadas simetricamente com uma chave que seu destinatário compartilha com você, eles poderiam facilmente ter forjado essas mensagens. Em contraste, com o PGP, as mensagens são assinadas e, portanto, o inverso da negabilidade — as mensagens são não repudiáveis. Até onde sei, nenhuma dessas propriedades foi efetivamente testada em tribunal.

Em 2010, o aplicativo móvel Signal (então chamado TextSecure) foi lançado, fazendo uso de um protocolo recém-criado chamado *Signal protocol*. Na época, a maioria dos protocolos de comunicação segura como PGP, S/MIME e OTR eram baseados em protocolos federados, onde nenhuma entidade central era necessária para que a rede funcionasse. O aplicativo Signal se afastou dessa tradição ao operar um serviço central e oferecer um único aplicativo oficial do Signal.

Embora o Signal impeça a interoperabilidade com outros servidores, o protocolo Signal é um padrão aberto e foi adotado por muitos outros aplicativos de mensagens, incluindo Google Allo (agora extinto), WhatsApp, Facebook Messenger, Skype, entre outros. O protocolo Signal é, de fato, uma história de sucesso, sendo usado de forma transparente por bilhões de pessoas, incluindo jornalistas, alvos de vigilância governamental e até minha avó de 92 anos (juro que não a forcei a instalar).

É interessante ver como o Signal funciona porque tenta corrigir muitas das falhas que mencionei anteriormente com o PGP. Nesta seção, abordarei cada uma das seguintes características interessantes do Signal:

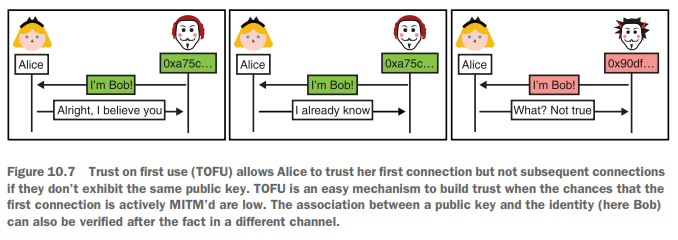
* Como podemos fazer melhor que a WOT? Existe uma maneira de atualizar os gráficos sociais existentes com criptografia de ponta a ponta? A resposta do Signal é usar uma abordagem de *trust on first use* (TOFU). O TOFU permite que os usuários confiem cegamente uns nos outros na primeira vez que se comunicam, confiando nessa primeira troca insegura para estabelecer um canal de comunicação seguro duradouro. Os usuários, então, podem verificar se a primeira troca foi vítima de MITM conferindo seu segredo de sessão fora de banda a qualquer momento no futuro.
* Como podemos atualizar o PGP para obter *forward secrecy* sempre que iniciarmos uma conversa com alguém? A primeira parte do protocolo Signal é como a maioria dos protocolos de transporte seguro: uma troca de chaves, mas uma troca particular chamada *Extended Triple Diffie-Hellman* (X3DH).
* Como podemos atualizar o PGP para obter *forward secrecy* para cada mensagem individual? Isso é importante porque conversas entre usuários podem durar anos, e um comprometimento em algum momento não deveria revelar anos de comunicação. O Signal aborda isso com algo chamado *symmetric ratchet*.
* E se os segredos de sessão de dois usuários forem comprometidos em algum momento? É o fim? Podemos nos recuperar disso? O Signal introduz uma nova propriedade de segurança chamada *post-compromise security* (PCS) e resolve isso com o que é chamado de *Diffie-Hellman (DH) ratchet*.

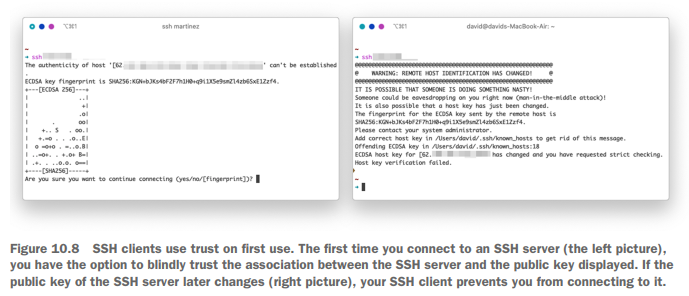
Vamos começar! Primeiro, veremos como o TOFU do Signal funciona.

**10.4.1 Mais amigável que a WOT: Confiar, mas verificar**

Uma das maiores falhas da criptografia de e-mail foi sua dependência do PGP e do modelo WOT para transformar gráficos sociais em gráficos sociais seguros. O design original do PGP previa que as pessoas se encontrassem pessoalmente para realizar uma cerimônia de assinatura de chaves (também chamada de *key-signing party*) para confirmar as chaves uns dos outros, mas isso era trabalhoso e inconveniente de várias maneiras. Hoje é realmente raro ver pessoas assinando as chaves PGP umas das outras.

A forma como a maioria das pessoas usa aplicativos como PGP, OTR, Signal e outros é confiar cegamente em uma chave na primeira vez que a vêem e rejeitar quaisquer alterações futuras (como ilustrado na figura 10.7). Assim, apenas a primeira conexão pode ser atacada (e isso apenas por um atacante MITM ativo).

Embora o TOFU não seja o melhor modelo de segurança, ele costuma ser o melhor que temos e tem se mostrado extremamente útil. O protocolo Secure Shell (SSH), por exemplo, é frequentemente usado confiando na chave pública do servidor durante a conexão inicial (ver figura 10.8) e rejeitando quaisquer alterações futuras.

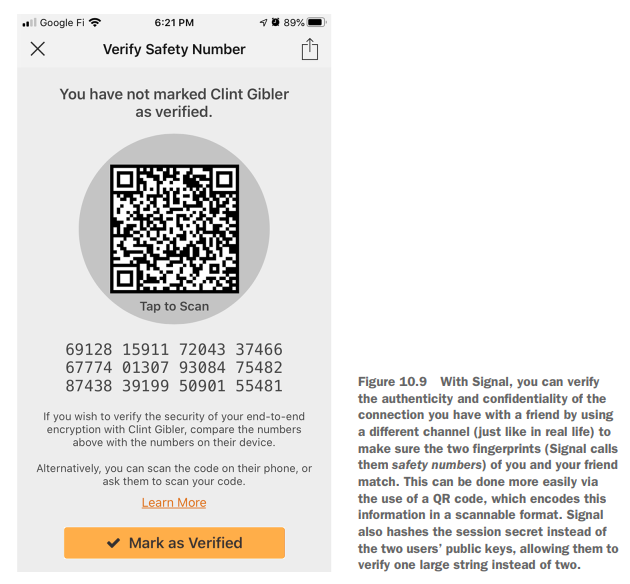
Enquanto sistemas TOFU confiam na primeira chave que veem, eles ainda permitem que o usuário verifique posteriormente se a chave é, de fato, a correta, e detectar tentativas de personificação. Em aplicativos do mundo real, os usuários normalmente comparam *fingerprints*, que são geralmente representações hexadecimais de chaves públicas ou de seus *hashes*. Essa verificação é, claro, feita fora de banda. (Se a conexão SSH estiver comprometida, então a verificação também estará.)

**NOTA** Claro, se os usuários não verificarem as *fingerprints*, poderão ser vítimas de MITM sem saber. Mas esse é o tipo de compromisso com o qual aplicativos do mundo real têm de lidar para levar a criptografia de ponta a ponta a larga escala. De fato, o fracasso da WOT mostra que aplicativos focados em segurança devem manter a usabilidade em mente para serem amplamente adotados.

No aplicativo Signal, uma *fingerprint* entre Alice e Bob é calculada por:

1. *Hash* da chave de identidade de Alice prefixada por seu nome de usuário (um número de telefone no Signal), interpretando uma truncagem desse *digest* como uma série de números;
2. O mesmo é feito para Bob;
3. Exibição da concatenação das duas séries de números para o usuário.

Aplicativos como o Signal utilizam códigos QR para facilitar a verificação das *fingerprints*, já que essas sequências podem ser longas. A figura 10.9 ilustra esse caso de uso.



**10.4.2 X3DH: O *handshake* do protocolo Signal**

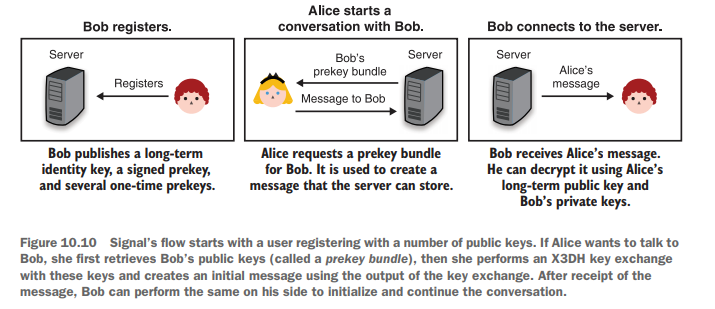
A maioria dos aplicativos de mensagens seguras antes do Signal eram síncronos. Isso significava que, por exemplo, Alice não poderia iniciar (ou continuar) uma conversa criptografada de ponta a ponta com Bob se Bob não estivesse online. O protocolo Signal, por outro lado, é assíncrono (como o e-mail), o que permite que Alice inicie (e continue) uma conversa com pessoas que estão offline.

Lembre-se de que *forward secrecy* (abordado no capítulo 9) significa que o comprometimento de chaves não compromete sessões anteriores, e que a *forward secrecy* normalmente implica em trocas de chaves interativas, pois ambos os lados precisam gerar pares de chaves efêmeras Diffie-Hellman (DH). Nesta seção, você verá como o Signal utiliza trocas de chaves não interativas (em que um dos lados pode estar offline) que ainda são seguras a longo prazo.

Para iniciar uma conversa com Bob, Alice inicia uma troca de chaves com ele. A troca de chaves do Signal, chamada X3DH, combina três (ou mais) trocas de chaves DH em uma só. Mas antes de entender como isso funciona, você precisa compreender os três diferentes tipos de chaves DH que o Signal utiliza:

* **Chaves de identidade** — São as chaves de longo prazo que representam os usuários. Você pode imaginar que, se o Signal usasse apenas chaves de identidade, o esquema seria bastante semelhante ao PGP, e não haveria *forward secrecy*.
* **Prekeys de uso único (one-time prekeys)** — Para adicionar *forward secrecy* à troca de chaves, mesmo quando o destinatário de uma nova conversa não está online, o Signal faz com que os usuários enviem antecipadamente múltiplas chaves públicas de uso único. São simplesmente chaves efêmeras que são carregadas previamente e são deletadas após serem usadas.
* **Prekeys assinadas (signed prekeys)** — Poderíamos parar aqui, mas ainda falta um caso limite. Como os *prekeys* de uso único podem, em algum momento, acabar, os usuários também devem enviar uma chave pública de médio prazo que eles assinam: um *signed prekey*. Assim, se não houver mais *prekeys* de uso único disponíveis no servidor sob seu nome de usuário, alguém ainda pode usar seu *signed prekey* para adicionar *forward secrecy* até o último momento em que você o atualizou. Isso significa que você precisa rotacionar periodicamente seu *signed prekey* (por exemplo, semanalmente).

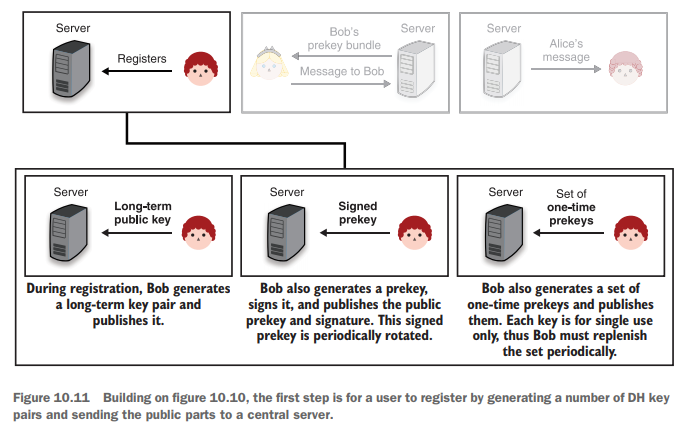
Isso já é o suficiente para termos uma visão geral de como o fluxo de criação de conversas no Signal funciona. A figura 10.10 apresenta uma visão geral.

Vamos detalhar cada um desses passos:

Primeiro, um usuário registra-se enviando:

* Uma chave de identidade;
* Um *signed prekey* e sua assinatura;
* Um número definido de *prekeys* de uso único.

Neste ponto, é responsabilidade do usuário rotacionar periodicamente o *signed prekey* e enviar novos *prekeys* de uso único. Eu recapitulo esse fluxo na figura 10.11.

**NOTA** O Signal utiliza a chave de identidade para realizar assinaturas sobre os *signed prekeys* e trocas de chaves durante o X3DH. Embora eu tenha alertado contra o uso da mesma chave para diferentes propósitos, o Signal analisou deliberadamente que, em seu caso, não há problema. Isso não significa que funcionaria no seu caso e com seu algoritmo de troca de chaves. Eu recomendaria, em geral, não reutilizar uma chave para múltiplos propósitos.

Após o passo da figura 10.11, Alice (voltando ao nosso exemplo) iniciaria uma conversa com Bob recuperando:

* A chave de identidade de Bob;
* O *signed prekey* atual de Bob e sua assinatura associada;
* Se ainda houver, um dos *prekeys* de uso único de Bob (o servidor então deleta o *prekey* de uso único enviado a Alice).

Alice pode verificar que a assinatura sobre o *signed prekey* está correta. Ela então realiza o *handshake* X3DH com:

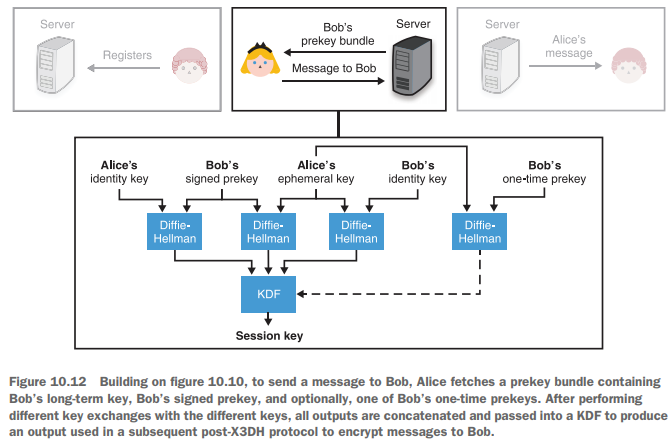
* Todas as chaves públicas de Bob;
* Um par de chaves efêmeras que ela gera na ocasião para adicionar *forward secrecy*;
* Sua própria chave de identidade.

A saída do X3DH é então usada em um protocolo pós-X3DH, que serve para criptografar suas mensagens para Bob (mais sobre isso na próxima seção). O X3DH é composto por três (opcionalmente quatro) trocas de chaves DH, agrupadas em uma. As trocas DH são realizadas entre:

1. A chave de identidade de Alice e o *signed prekey* de Bob;
2. A chave efêmera de Alice e a chave de identidade de Bob;
3. A chave efêmera de Alice e o *signed prekey* de Bob;
4. Se Bob ainda tiver um *prekey* de uso único disponível, seu *prekey* e a chave efêmera de Alice.

A saída do X3DH é a concatenação de todas essas trocas DH, passada a uma função de derivação de chave (*key derivation function* — KDF), abordada no capítulo 8. Diferentes trocas de chaves fornecem propriedades diferentes. As duas primeiras são para autenticação mútua, enquanto as duas últimas são para *forward secrecy*. Tudo isso é analisado com mais profundidade na especificação X3DH (<https://signal.org/docs/specifications/x3dh/>), que recomendo a leitura.

A figura 10.12 recapitula esse fluxo.

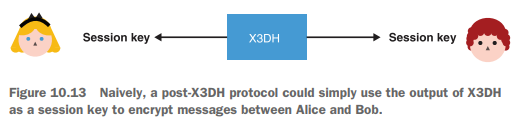
Alice agora pode enviar a Bob sua chave pública de identidade, a chave pública efêmera que gerou para iniciar a conversa e outras informações relevantes (como qual *prekey* de uso único de Bob foi usado). Bob recebe a mensagem e pode realizar exatamente o mesmo X3DH com as chaves públicas contidas nela. (Por essa razão, eu não ilustro o último passo desse fluxo.) Se Alice usou um *prekey* de uso único de Bob, Bob o descarta.

Agora, o que acontece depois que o X3DH é concluído? Vamos ver isso a seguir.

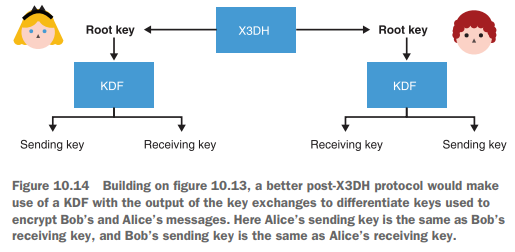
**10.4.3 *Double Ratchet*: o protocolo pós-*handshake* do Signal**

A fase pós-X3DH dura enquanto os dois usuários não deletarem suas conversas ou perderem qualquer uma de suas chaves. Por essa razão — e porque o Signal foi projetado com conversas via SMS em mente, nas quais o intervalo entre duas mensagens pode ser de meses — o Signal introduz *forward secrecy* no nível de cada mensagem. Nesta seção, você aprenderá como esse protocolo pós-*handshake* (chamado *Double Ratchet*) funciona.

Mas primeiro, imagine um protocolo pós-X3DH simples. Alice e Bob poderiam ter usado a saída do X3DH como uma chave de sessão e usá-la para criptografar mensagens entre eles, como ilustrado na figura 10.13.

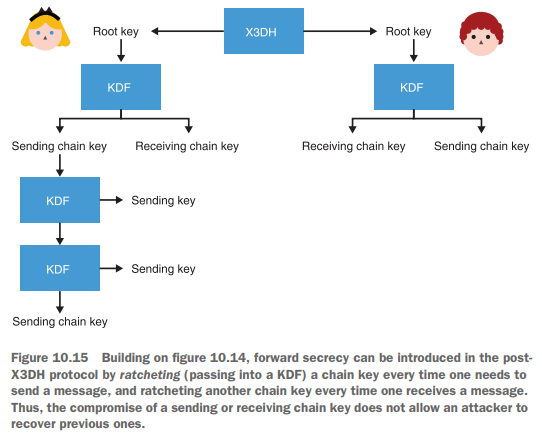


Geralmente, desejamos separar as chaves utilizadas para diferentes propósitos. O que podemos fazer é usar a saída do X3DH como uma semente (ou chave raiz, segundo a especificação *Double Ratchet*) para uma função de derivação de chave (KDF) a fim de derivar duas outras chaves. Alice pode usar uma chave para criptografar mensagens para Bob, e Bob pode usar a outra chave para criptografar mensagens para Alice. Eu ilustro isso na figura 10.14.



Essa abordagem poderia ser suficiente, mas o Signal observa que sessões de mensagens podem durar anos — ao contrário das sessões TLS do capítulo 9, que geralmente são de curta duração. Por causa disso, se a chave de sessão for comprometida em qualquer ponto no tempo, todas as mensagens previamente gravadas poderão ser descriptografadas!

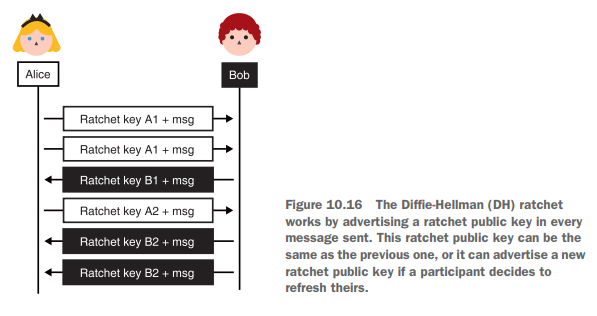
Para corrigir isso, o Signal introduziu o que chama de *symmetric ratchet* (como ilustrado na figura 10.15). A chave de envio agora é renomeada como *sending chain key* e não é usada diretamente para criptografar mensagens. Ao enviar uma mensagem, Alice continuamente passa essa *sending chain key* em uma função unidirecional, que produz a próxima *sending chain key* bem como a chave de envio efetiva para criptografar a mensagem. Bob, por outro lado, faz o mesmo com a *receiving chain key*. Assim, ao comprometer uma *sending key* ou uma *sending chain key*, um atacante não pode recuperar chaves anteriores. (O mesmo vale ao receber mensagens.)



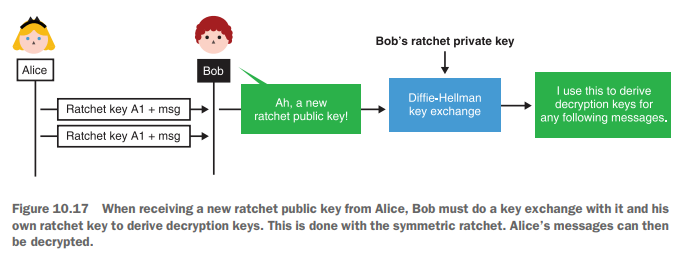
Agora temos *forward secrecy* integrada ao nosso protocolo e em nível de mensagem. Cada mensagem enviada e recebida protege todas as mensagens enviadas e recebidas anteriormente. Note que isso é algo debatível, pois um atacante que comprometa uma chave provavelmente faz isso comprometendo o telefone do usuário, o qual muito provavelmente contém todas as mensagens anteriores em texto claro ao lado da chave. Ainda assim, se ambos os usuários em uma conversa decidirem apagar mensagens anteriores (por exemplo, usando o recurso de “mensagens que desaparecem” do Signal), a propriedade de *forward secrecy* é alcançada.

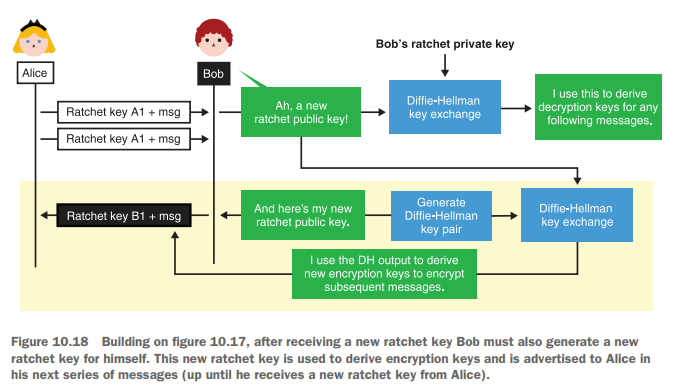
O protocolo Signal ainda possui mais uma característica interessante: o *post-compromise security* (PCS), também chamado de *backward secrecy*, como você aprendeu no capítulo 8. O PCS é a ideia de que, se suas chaves forem comprometidas em algum momento, você ainda pode se recuperar, pois o protocolo se autorrepara. Claro, se o atacante ainda tiver acesso ao seu dispositivo após o comprometimento, então tudo isso será inútil.

O PCS só funciona ao reintroduzir nova entropia à qual um comprometimento não persistente não teria acesso. A nova entropia deve ser a mesma para ambos os pares. A forma como o Signal obtém essa entropia é realizando uma nova troca de chaves efêmera. Para isso, o protocolo Signal continuamente realiza trocas de chaves no que é chamado de *DH ratchet*. Cada mensagem enviada pelo protocolo vem com a chave pública atual do *ratchet*, como ilustrado na figura 10.16.



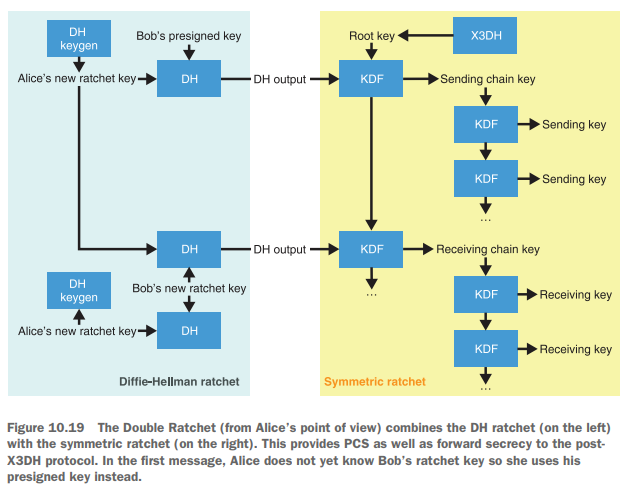
Quando Bob percebe uma nova chave *ratchet* de Alice, ele deve realizar uma nova troca de chaves DH com a nova chave *ratchet* de Alice e a sua própria chave *ratchet*. A saída pode então ser usada com o *symmetric ratchet* para descriptografar as mensagens recebidas. Eu ilustro isso na figura 10.17.

Outro passo que Bob deve tomar ao receber uma nova chave *ratchet* é gerar uma nova chave *ratchet* para si mesmo. Com essa nova chave *ratchet*, ele pode realizar outra troca de chaves com a nova chave *ratchet* de Alice, que ele então usa para criptografar mensagens para ela. Isso deve se parecer com a figura 10.18.

Essa troca contínua de chaves é mencionada como um “ping-pong” na especificação *Double Ratchet*:

Isso resulta em um comportamento de "ping-pong", enquanto as partes alternam a substituição de pares de chaves *ratchet*. Um espião que comprometer temporariamente uma das partes pode aprender o valor de uma chave privada atual do *ratchet*, mas essa chave privada será eventualmente substituída por uma não comprometida. Nesse ponto, o cálculo Diffie-Hellman entre os pares de chaves *ratchet* definirá uma saída DH desconhecida para o atacante.  
— *The Double Ratchet Algorithm*

Finalmente, a combinação do *DH ratchet* e do *symmetric ratchet* é chamada de *Double Ratchet*. É um pouco denso visualizar tudo em um único diagrama, mas a figura 10.19 tenta ilustrar isso.



Sei que esse último diagrama é bastante denso, por isso incentivo você a consultar as especificações do Signal, publicadas em <https://signal.org/docs>. Elas fornecem outra explicação bem escrita do protocolo.

**10.5 O estado da criptografia de ponta a ponta**

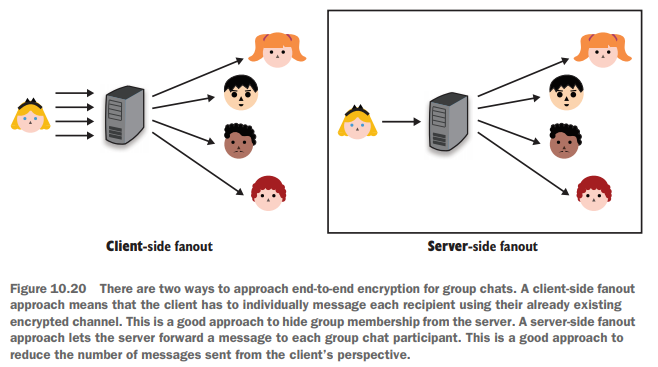
Hoje, a maioria das comunicações seguras entre usuários acontece por meio de aplicativos de mensagens seguras em vez de e-mails criptografados. O protocolo Signal tem sido o claro vencedor em sua categoria, sendo adotado por muitos aplicativos proprietários e também por protocolos abertos e federados como o XMPP (via a extensão OMEMO) e o Matrix (uma alternativa moderna ao IRC). Por outro lado, o PGP e o S/MIME estão sendo abandonados, à medida que ataques publicados resultaram em perda de confiança.

E se você quiser escrever seu próprio aplicativo de mensagens com criptografia de ponta a ponta? Infelizmente, muito do que está sendo usado neste campo é ad hoc, e você teria que preencher muitos dos detalhes por conta própria para obter um sistema completo e seguro. O Signal disponibilizou muito de seu código como *open source*, mas ele carece de documentação e pode ser difícil de usar corretamente. Por outro lado, você pode ter mais sucesso utilizando uma solução descentralizada *open source* como o Matrix, que pode ser mais fácil de integrar. Foi o que o governo francês fez.

Antes de encerrarmos este capítulo, há também uma série de questões em aberto e problemas de pesquisa ativa que quero abordar. Por exemplo:

* Mensagens em grupo
* Suporte a múltiplos dispositivos
* Garantias de segurança melhores do que o modelo TOFU

Vamos começar com o primeiro item: mensagens em grupo. Neste ponto, embora implementado de maneiras diferentes por diferentes aplicativos, as mensagens em grupo ainda estão sendo pesquisadas ativamente. Por exemplo, o aplicativo Signal faz com que os clientes lidem com os chats em grupo. Os servidores só veem pares de usuários conversando — nunca menos, nunca mais. Isso significa que os clientes precisam criptografar uma mensagem de chat em grupo para todos os participantes e enviá-las individualmente. Isso é chamado de *client-side fanout* e não escala muito bem. Também não é difícil para o servidor deduzir quem são os membros do grupo quando vê Alice, por exemplo, enviando várias mensagens de mesmo comprimento para Bob e Charles (figura 10.20).



O WhatsApp, por outro lado, usa uma variante do protocolo Signal onde o servidor tem conhecimento da composição dos grupos de chat. Essa mudança permite que um participante envie uma única mensagem criptografada ao servidor, que, por sua vez, ficará responsável por encaminhá-la aos membros do grupo. Isso é chamado de *server-side fanout*.

Outro problema dos chats em grupo é escalar para grupos com um número grande de membros. Para isso, muitos atores da indústria recentemente se reuniram em torno do padrão *Messaging Layer Security* (MLS) para lidar com mensagens seguras em grupo em grande escala. Mas parece haver muito trabalho a ser feito, e cabe se perguntar: há realmente confidencialidade em um chat em grupo com mais de cem participantes?

**NOTA** Este ainda é um campo de pesquisa ativa, e diferentes abordagens vêm com diferentes compromissos entre segurança e usabilidade. Por exemplo, em 2021, nenhum protocolo de chat em grupo parece fornecer *transcript consistency*, uma propriedade criptográfica que garante que todos os participantes de um chat em grupo vejam as mesmas mensagens na mesma ordem.

O suporte a múltiplos dispositivos ou não existe ou é implementado de várias maneiras, muitas vezes fingindo que seus diferentes dispositivos são participantes distintos de um chat em grupo. O modelo TOFU pode tornar o manuseio de múltiplos dispositivos bastante complicado porque ter diferentes chaves de identidade por dispositivo pode se tornar um verdadeiro problema de gerenciamento de chaves. Imagine ter que verificar as *fingerprints* de cada um de seus dispositivos e de cada dispositivo dos seus amigos. O Matrix, por exemplo, faz com que o usuário assine seus próprios dispositivos. Outros usuários podem então confiar em todos os seus dispositivos como uma única entidade ao verificar suas assinaturas associadas.

Por fim, mencionei que o modelo TOFU também não é o ideal, pois baseia-se em confiar em uma chave pública na primeira vez em que a vemos, e a maioria dos usuários não verifica posteriormente se as *fingerprints* coincidem. Algo pode ser feito a respeito? E se o servidor decidir se passar por Bob apenas para Alice? Esse é um problema que o *Key Transparency* está tentando resolver. O *Key Transparency* é um protocolo proposto pelo Google, semelhante ao *Certificate Transparency* tratado no capítulo 9. Há também algumas pesquisas que fazem uso da tecnologia *blockchain*, sobre a qual falarei no capítulo 12 sobre criptomoedas.

**Resumo**

* A criptografia de ponta a ponta é sobre proteger comunicações entre seres humanos reais. Um protocolo que implementa criptografia de ponta a ponta é mais resiliente a vulnerabilidades que possam ocorrer em servidores que ficam entre os usuários e pode simplificar muito as exigências legais para empresas.
* Sistemas de criptografia de ponta a ponta precisam de uma forma de iniciar a confiança entre os usuários. Essa confiança pode vir de uma chave pública já conhecida ou de um canal fora de banda confiável.
* PGP e S/MIME são os principais protocolos usados atualmente para criptografar e-mails, mas nenhum deles é considerado seguro, pois utilizam algoritmos e práticas criptográficas antigas. Eles também têm integração ruim com clientes de e-mail, que já se mostraram vulneráveis a vários ataques na prática.
  + O PGP usa o modelo *web of trust* (WOT), onde usuários assinam as chaves públicas uns dos outros para permitir que outros possam confiar nelas.
  + O S/MIME utiliza uma infraestrutura de chave pública (PKI) para construir confiança entre participantes. É mais comumente usado em empresas e universidades.
* Uma alternativa ao PGP é o *saltpack*, que corrige vários problemas enquanto confia em redes sociais para descoberta de chaves públicas de outras pessoas.
* Os e-mails sempre terão problemas com criptografia, pois o protocolo foi construído sem considerar criptografia desde o início. Por outro lado, protocolos e aplicativos modernos de mensagens são melhores alternativas para e-mails criptografados, já que são construídos com criptografia de ponta a ponta em mente.
  + O protocolo Signal é usado pela maioria dos aplicativos de mensagens para proteger comunicações de ponta a ponta entre usuários. Signal Messenger, WhatsApp, Facebook Messenger e Skype todos divulgam o uso do protocolo Signal para proteger as mensagens.
  + Outros protocolos, como o Matrix, tentam padronizar protocolos federados para mensagens criptografadas de ponta a ponta. Protocolos federados são protocolos abertos com os quais qualquer um pode interoperar (ao contrário de protocolos centralizados, que se limitam a um único aplicativo).